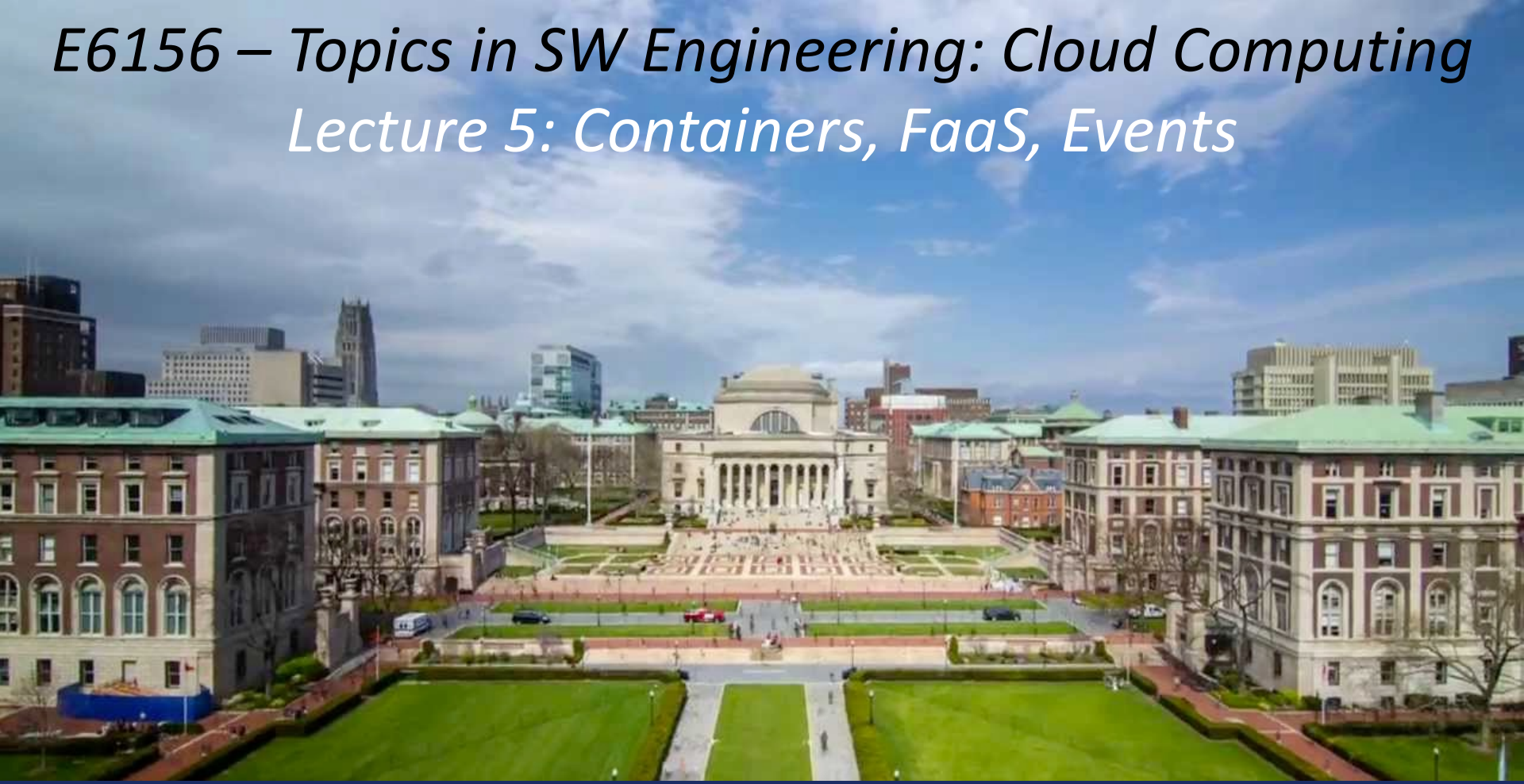
E6156 – Topics in SW Engineering: Cloud Computing

Lecture 5: Containers, FaaS, Events

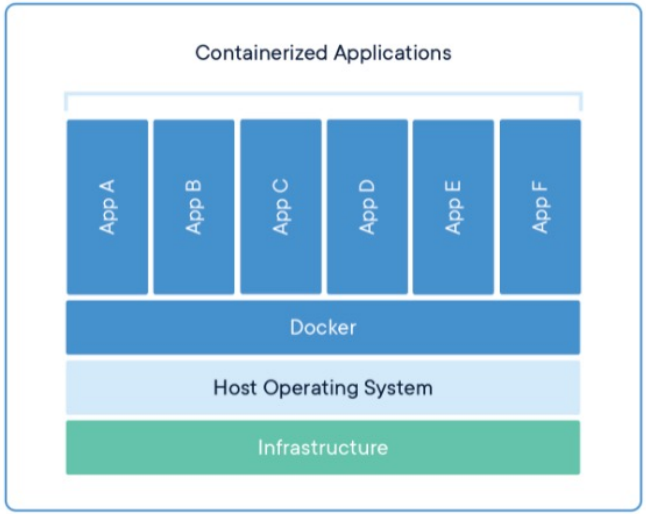


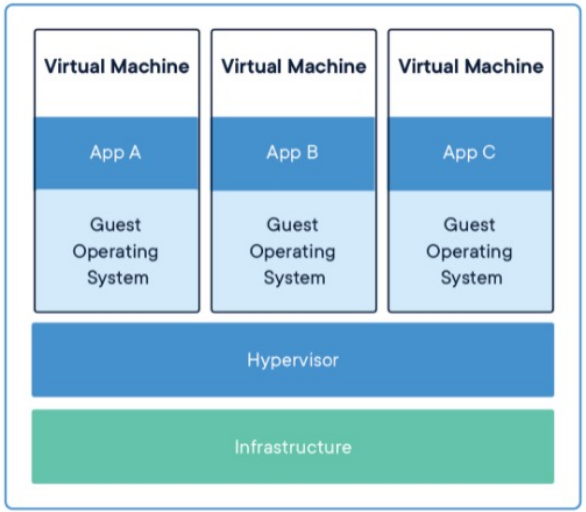
Containers

Fundamental, Recurring Problem

- I have one big computer and I want to run multiple applications.
- This creates several challenges, e.g.
  - Resource sharing and allocation (CPU, memory, disk, ...)
  - Isolation: Application A should not be able to corrupt/mess with application B's files, memory, ...
  - Configuration: Applications require libraries, versions of libraries, prerequisites, compiler/interpreter levels, ...
- Basically, I want a way to someone package or fence all of this.
  We see some of the issues with Java Classpath, Python environments, ...
- There are a few ways to do this:
  - OS process and paths.
  - VMs
  - Containers

Containers and VMs





CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Concept (from Wikipedia)

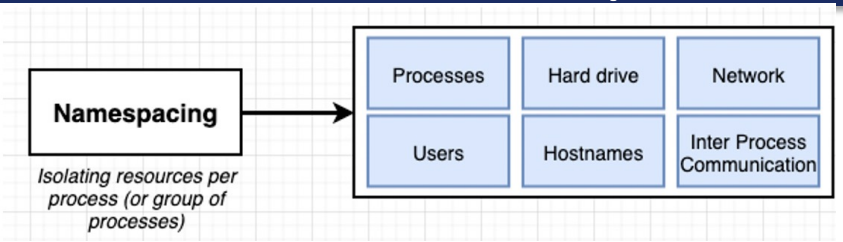
- “cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.”

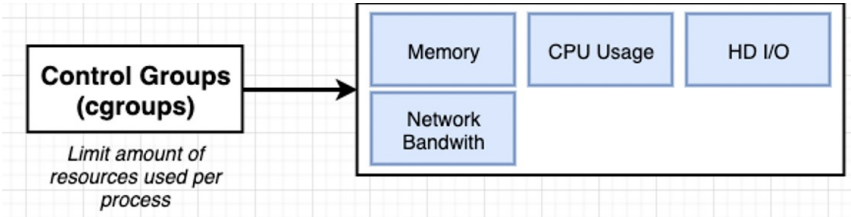
- “Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.
  … …
  those namespaces refer to distinct resources. Resources may exist in multiple spaces. Examples of such resources are process IDs, hostnames, user IDs, file names, and some names associated with network access, ...”

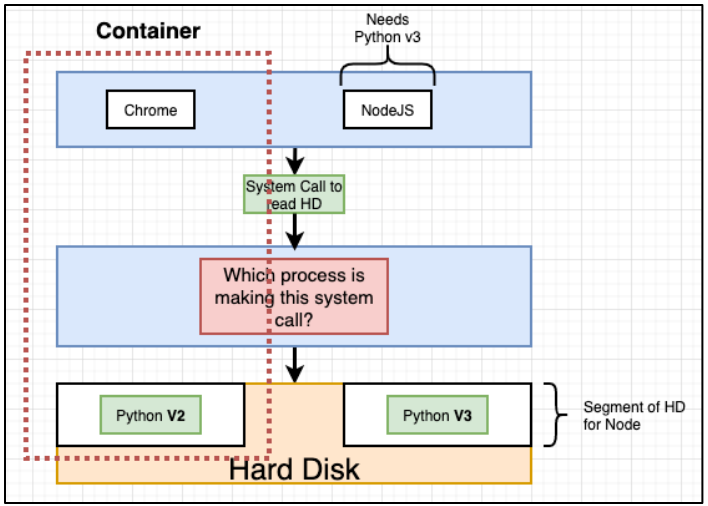
Containers vs VMs

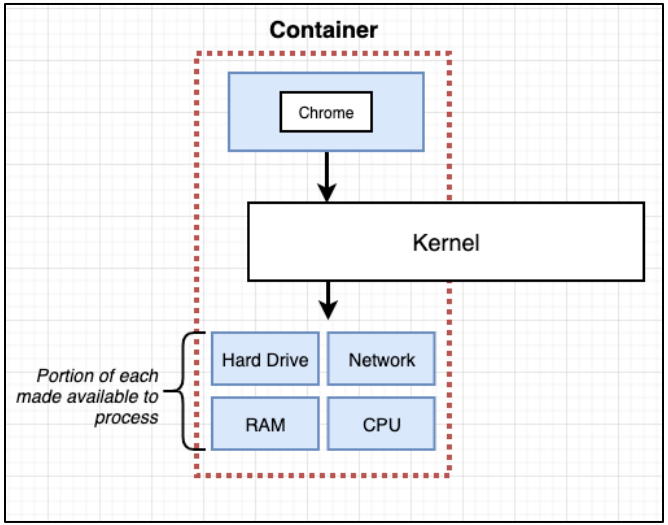
	Process	Container	VM
Definition	A representation of a running program.	Isolated group of processes managed by a shared kernel.	A full OS that shares host hardware via a hypervisor.
Use case	Abstraction to store state about a running process.	Creates isolated environments to run many apps.	Creates isolated environments to run many apps.
Type of OS	Same OS and distro as host,	Same kernel, but different distribution.	Multiple independent operating systems.
OS isolation	Memory space and user privilieges.	Namespaces and cgroups.	Full OS isolation.
Size	Whatever user's application uses.	Images measured in MB + user's application.	Images measured in GB + user's application.
Lifecycle	Created by forking, can be long or short lived, more often short.	Runs directly on kernel with no boot process, often is short lived.	Has a boot process and is typically long lived.

Some Container/Docker Implementation Concepts





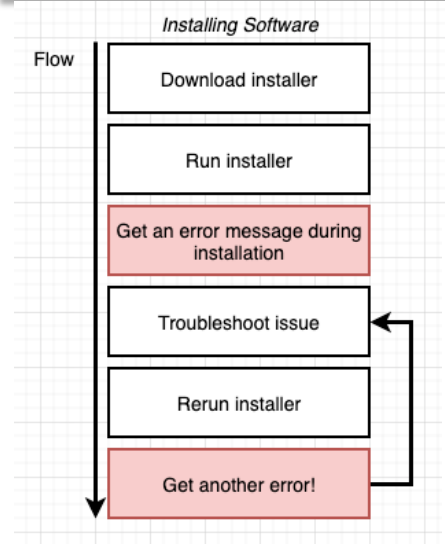




Isolates libraries, paths, packages, PIDs, ...

Partitions and caps resource consumption.

Containers and Some Docker Overview



This is what Docker is trying to fix

Why use Docker? → Docker makes it really easy to install and run software without worrying about setup or dependencies

- We are familiar with the problem
  - We have a system with an OS and installed SW.
  - We need to install something new, and that requires installing prerequisites, changing versions, etc.
  - After many tries, we get the application installed only to find out we broke some other application.
- Docker and containers allow you to:
  - Develop an application or SW system.
  - Package up the entire environment (paths, versions, libs, ...) into an image that works and is self-contained.
  - Someone installing/starting your application simply gets the image from a repository and starts the image to create a running container.

Simple Tutorial

- We will follow this tutorial https://docs.docker.com/language/python/ to get experience.
- But, the in the future, basic idea is the same as we have previously followed.
- The steps:
  - Find an example of something cool.
  - Download it, set it up and run it.
  - Modify it and add my application code and logic.
  - Push it somewhere, in this case Docker Hub
  - Deploy on the cloud. AWS gives you two or three options:
    - Docker on EC2
    - Elastic Container Service
- I also used this: https://www.docker.com/blog/containerized-python-development-part-2/

A Little More REST

REST Method

- When your handler receives a REST operation, there are the following elements. Not all methods have all elements:
  - Path
  - Path parameters
  - Query string parameters
  - Headers
  - Body
- The REST handler's response has the following elements:
  - Status code
  - Body
  - Headers
- The route handler maps between the REST representation of the elements and the language/runtime representation.

- Flask provides access through the
  - Request object.
  - Response object.
- Walkthrough

REST Paths https://restful-api-design.readthedocs.io/en/latest/

- There are many patterns. I typically follow something like:

URL Structure

Each collection and resource in the API has its own URL. URLs should never be constructed by an API client. Instead, the client should only follow links that are generated by the API itself.

The recommended convention for URLs is to use alternate collection / resource path segments, relative to the API entry point. This is best described by example. The table below uses the ":name" URL variable style from Rail's "Routes" implementation.

URL	Description
/api	The API entry point
/api/:coll	A top-level collection named "coll"
/api/:coll/:id	The resource "id" inside collection "coll"
/api/:coll/:id/:subcoll	Sub-collection "subcoll" under resource "id"
/api/:coll/:id/:subcoll/:subid	The resource "subid" inside "subcoll"

Even though sub-collections may be arbitrarily nested, in my experience, you want to keep the depth limited to 2, if possible. Longer URLs are more difficult to work with when using simple command-line tools like curl.

Method	Scope	Semantics
GET	collection	Retrieve all resources in a collection
GET	resource	Retrieve a single resource
HEAD	collection	Retrieve all resources in a collection (header only)
HEAD	resource	Retrieve a single resource (header only)
POST	collection	Create a new resource in a collection
PUT	resource	Update a resource
PATCH	resource	Update a resource
DELETE	resource	Delete a resource
OPTIONS	any	Return available HTTP methods and other options

- We will not be too rigorous about status codes, but some common examples are:
  - 200: OK (success)
  - 201: CREATED (for POST)
  - 404: NOT FOUND
  - 500: INTERNAL SERVICE ERROR
  - 418: I'M A TEAPOT

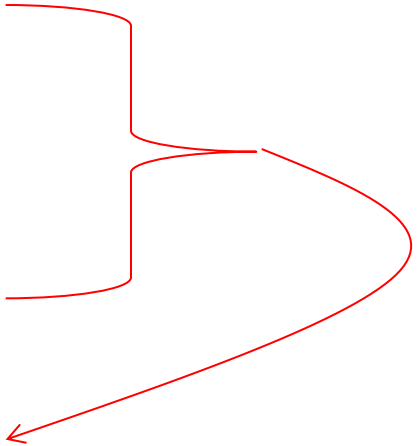
Success Response Codes

Operation	HTTP Request	HTTP Response Codes Supported
READ	GET	200 - OK with message body 204 - OK no message body 206 - OK with partial message body
CREATE	POST	201 - Resource created (Operation Complete) 202 - Resource accepted (Operation Pending)
UPDATE	PUT	202 - Accepted (Operation Pending) 204 - Success (Operation Complete)
DELETE	DELETE	202 - Accepted (Operation Pending) 204 - Success (Operation Complete)

202 means

Your request went asynch.

The HTTP header Link is where to poll for rsp.

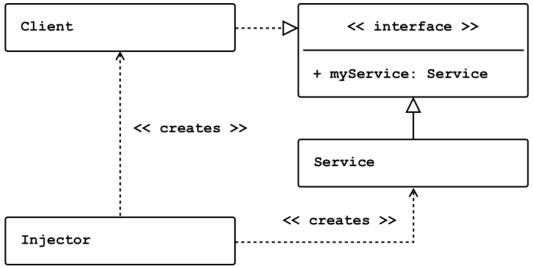


Examples of Link Headers in HTTP response:

```
Link: <http://api/jobs/j1>;rel=monitor;title="update profile"
Link: <http://api/reports/r1>;rel=summary;title="access report"
```

Dependency Injection

- Concepts (https://en.wikipedia.org/wiki/Dependency_injection):
  - In software engineering, dependency injection is a technique in which an object receives other objects that it depends on, called dependencies.
  - The intent behind dependency injection is to achieve separation of concerns of construction and use of objects. This can increase readability and code reuse.

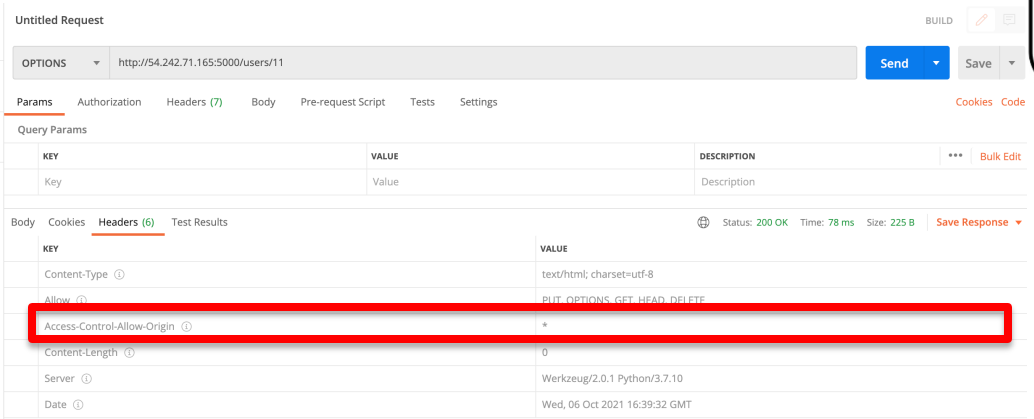


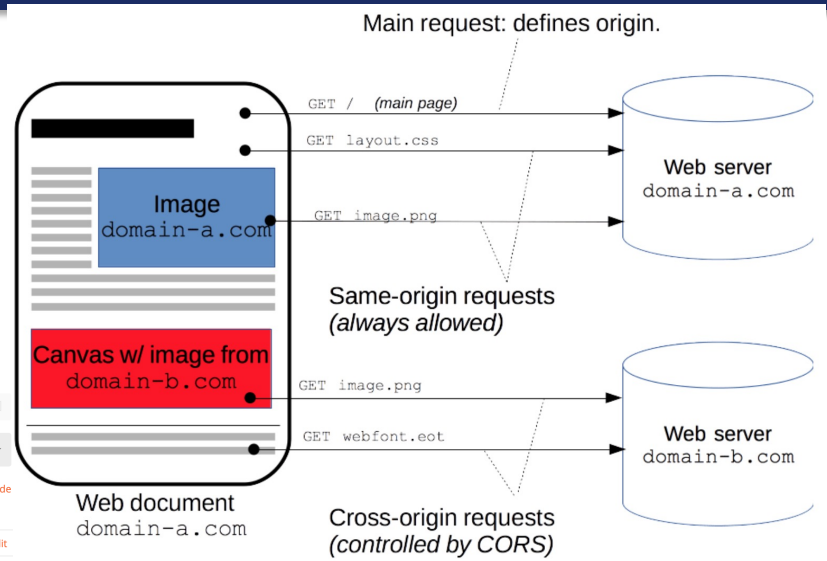

```
class UserResource(BaseApplicationResource):

    def __init__(self, config_info):
        super().__init__(config_info)
```

- There is no single approach. There are several frameworks, tools, etc.
- Like many other things, it is easy to get carried away and become dogmatic.
- I follow a single approach:
  - A Context class converts environment and other configuration and provides to application.
  - The top-level application injects a config_info object into services.

CORS https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

- "For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. For example, XMLHttpRequest and the Fetch API follow the same-origin policy. This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers."






```
app = Flask(__name__)
CORS(app)
```

REST Principles
HATEOAS

REST Principles

- REST Principles:
  - Uniform interface
  - Client–server
  - Stateless
  - Cacheable
  - Layered system
  - Code on demand (optional)

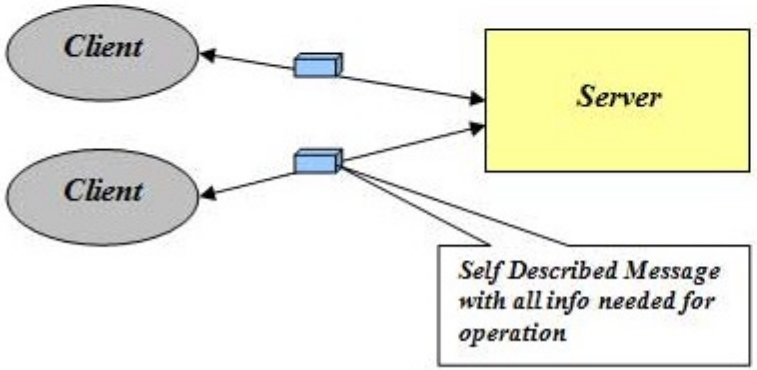
HATEOAS: Hypertext as the Engine of Application State -- The principle is that a client interacts with a network application entirely through hypermedia provided dynamically by application servers. A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

- Many of these are straightforward. We will explore some of the concepts in various lectures.

- Today, we cover “statelessness.”

Stateless

The notion of statelessness is defined from the perspective of the server. The constraint says that the server should not remember the state of the application. As a consequence, the client should send all information necessary for execution along with each request, because the server cannot reuse information from previous requests as it didn't memorize them. All info needed is in message.



Statelessness

2. Application State vs Resource State

It is important to understand the between the application state and the resource state. Both are completely different things.

Application state is server-side data that servers store to identify incoming client requests, their previous interaction details, and current context information.

Resource state is the current state of a resource on a server at any point in time – and it has nothing to do with the interaction between client and server. It is what we get as a response from the server as the API response. We refer to it as resource representation.

> REST statelessness means being free from the application state.

3. Advantages of Stateless APIs

There are some very noticeable advantages of having **REST APIs stateless**.

1. Statelessness helps in **scaling the APIs to millions of concurrent users** by deploying it to multiple servers. Any server can handle any request because there is no session related dependency.
2. Being stateless makes REST APIs **less complex** – by removing all server-side state synchronization logic.
3. A stateless API is also **easy to cache** as well. Specific softwares can decide whether or not to cache the result of an HTTP request just by looking at that one request. There's no nagging uncertainty that state from a previous request might affect the cacheability of this one. It **improves the performance** of applications.
4. The server never loses track of "where" each client is in the application because the client sends all necessary information with each request.

- This concept can be confusing with experience with other approach to session state.
- Some application frameworks did/do keep session state on the server.
- Application servers and frameworks have various support for "session state," e.g. https://flask-session.readthedocs.io/en/latest/

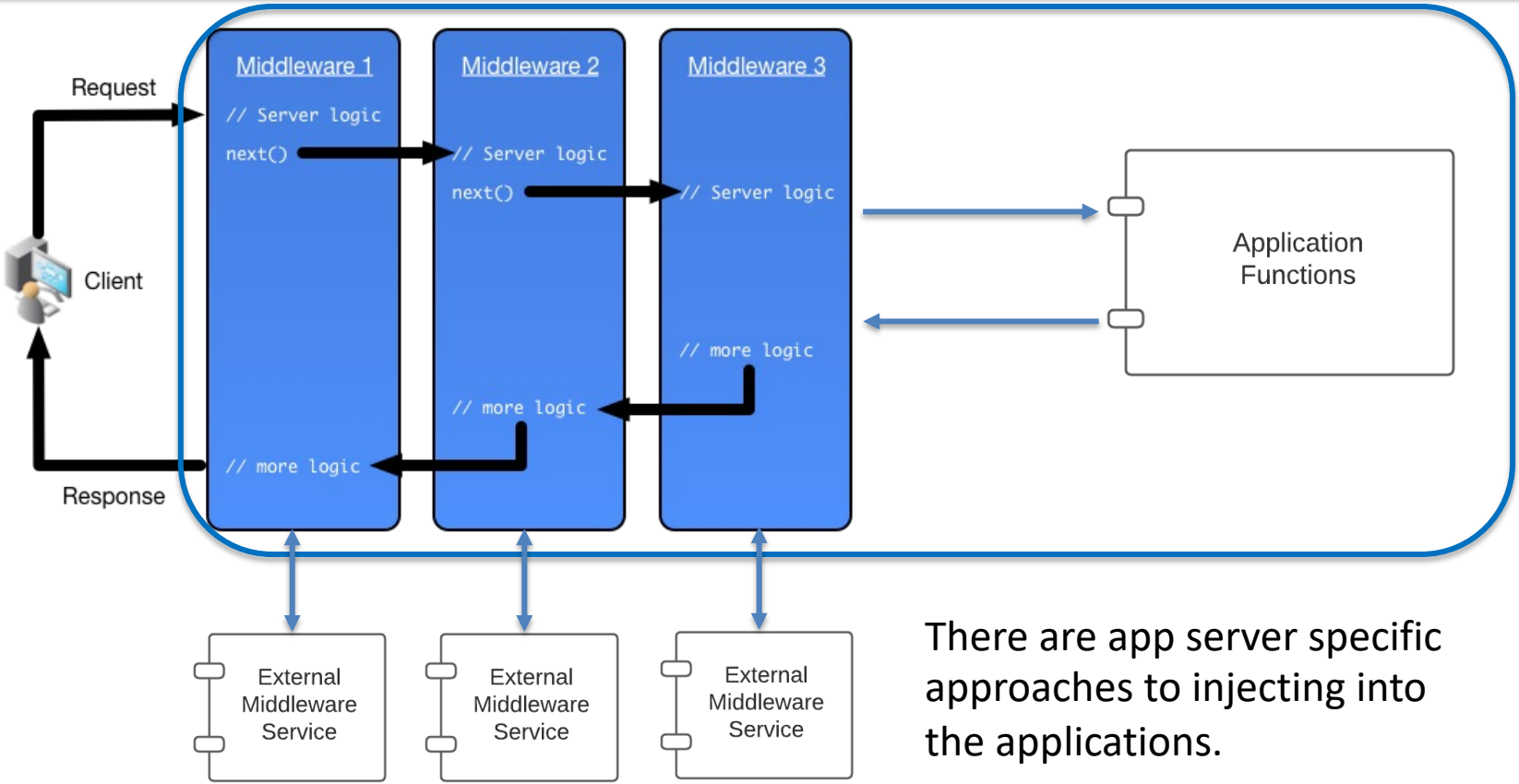
Middleware

What is middleware?

Middleware is software that lies between an operating system and the applications running on it. Essentially functioning as hidden translation layer, middleware enables communication and data management for distributed applications. It's sometimes called plumbing, as it connects two applications together so data and databases can be easily passed between the "pipe." Using middleware allows users to perform such requests as submitting forms on a web browser, or allowing the web server to return dynamic web pages based on a user's profile.

Common middleware examples include database middleware, application server middleware, message-oriented middleware, web middleware, and transaction-processing monitors. Each program typically provides messaging services so that different applications can communicate using messaging frameworks like simple object access protocol (SOAP), web services, representational state transfer (REST), and JavaScript object notation (JSON). While all middleware performs communication functions, the type a company chooses to use will depend on what service is being used and what type of information needs to be communicated. This can include security authentication, transaction management, message queues, applications servers, web servers, and directories. Middleware can also be used for distributed processing with actions occurring in real time rather than sending data back and forth.

Middleware – Conceptual Model



There are app server specific approaches to injecting into the applications.

Middleware and Flask

```python
# These methods get called before/after. You can check security information for all requests.
@application.before_request
def before_decorator():
    a_ok = sec.check_authetication(request)
    print("a_ok = ", a_ok)
    if a_ok[0] != 200:
        handle_error(a_ok[0], a_ok[1], a_ok[2])


@application.after_request
def after_decorator(rsp):
    print("... In after decorator ...")
    notify.notify(request, rsp)
    return rsp
```

- White list of URLs
- If not on white list:
  - Verify token
  - Verify authorization

- List of filters of the form:
  - URL
  - HTTP Method
  - SNS topic to use
- Send an event to SNS if matches filter

```python
@application.route("/api/users", methods=["GET", "POST"])
def users_get():...
```

Show some demo code.

So, How Will We Use This?

- Add before_request and after_request to microservices.
  (Note: There are other approaches).
- Implement two middleware interceptors:
  - Security
    - Configured with a JSON file.
    - File lists which {path, method} require being logged on.
    - While IDs can perform which methods on which paths.
    - Automates driving login redirect (401 – Unauthorized)
    - Checks authorization (403– Forbidden)
  - Notification:
    - Configured with JSON file.
    - File lists which operations emit a before and/or after event to an SNS topic.
- We will use notification to drive some other things like webhooks.

Event Drive Processing
Publish/Subscribe

Pub/Sub – Event Driven Processing

"In software architecture, **publish–subscribe** is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

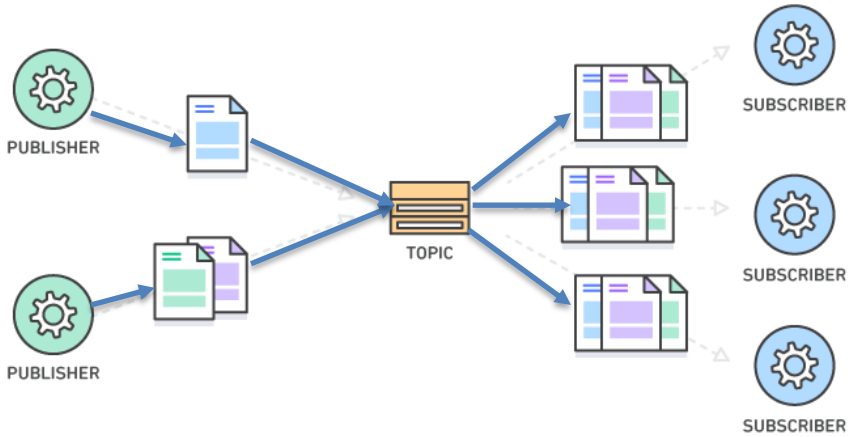
Publish–subscribe is a sibling of the message queue paradigm, and is typically one part of a larger message-oriented middleware system. Most messaging systems support both the pub/sub and message queue models in their API, e.g. Java Message Service (JMS).

This pattern provides greater network scalability and a more dynamic network topology, with a resulting decreased flexibility to modify the publisher and the structure of the published data."

https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

It also support loose coupling for application evolution and agility.

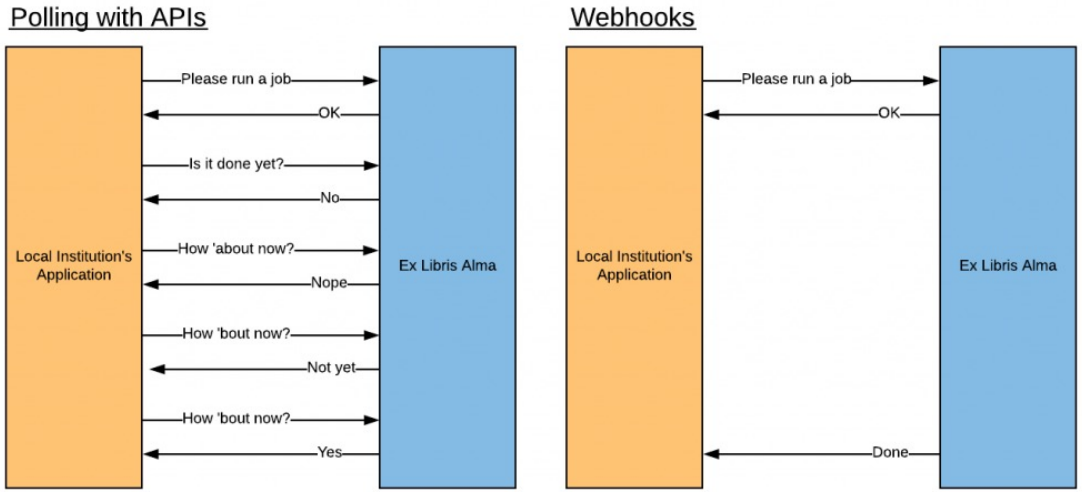
I can add and evolve services without modifying the services.



Pub/Sub – Simple Notification Service

Webhooks

- “A webhook in web development is a method of augmenting or altering the behavior of a web page or web application with custom callbacks. These callbacks may be maintained, modified, and managed by third-party users and developers who may not necessarily be affiliated with the originating website or application. The term "webhook" was coined by Jeff Lindsay in 2007 from the computer programming term hook.”



Function-as-a-Service

Serverless and Function-as-a-Service

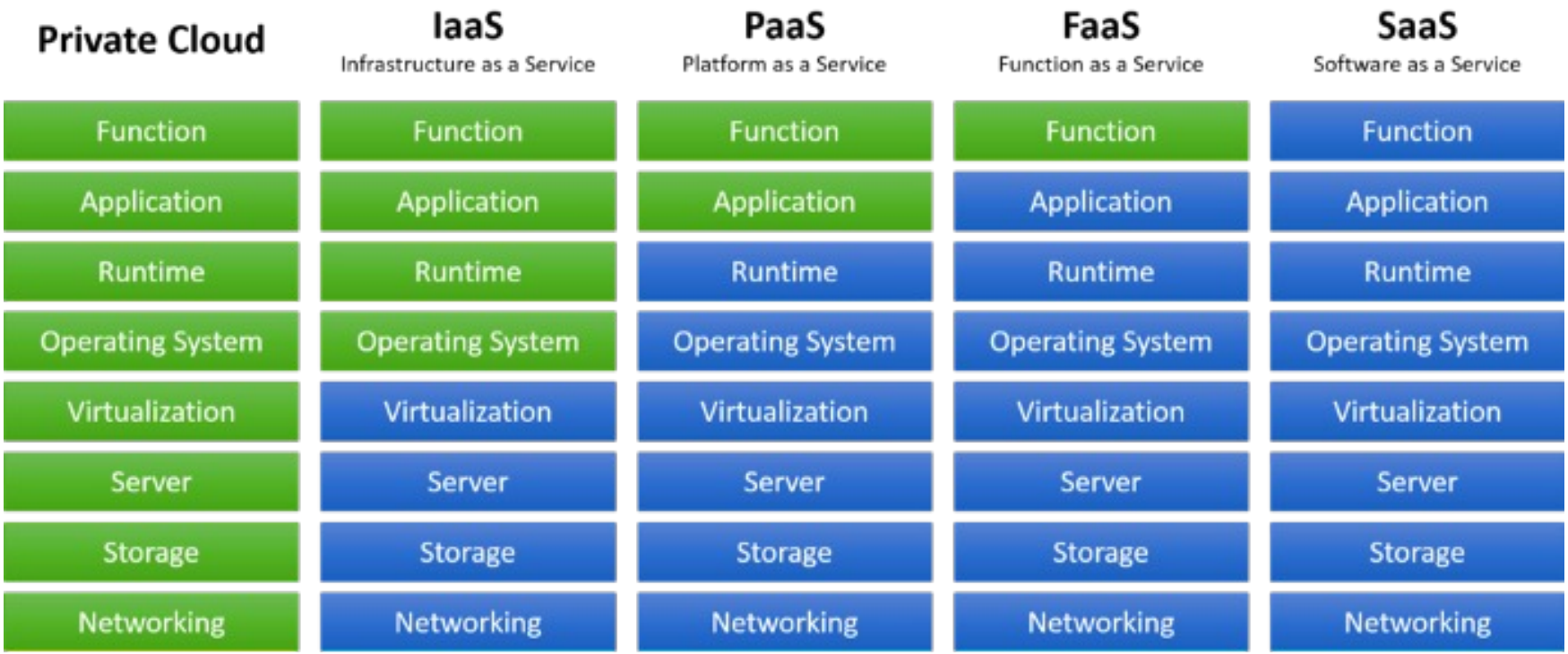
- **Serverless computing** is a cloud computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of machine resources. Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity.[1] It can be a form of utility computing.

  Serverless computing can simplify the process of deploying code into production. Scaling, capacity planning and maintenance operations may be hidden from the developer or operator. Serverless code can be used in conjunction with code deployed in traditional styles, such as microservices. Alternatively, applications can be written to be purely serverless and use no provisioned servers at all.[2] This should not be confused with computing or networking models that do not require an actual server to function, such as peer-to-peer (P2P)."

- **Function as a service** (**FaaS**) is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.[1] Building an application following this model is one way of achieving a "serverless" architecture, and is typically used when building microservices applications."

- That was baffling:
  - IaaS, CaaS – You control the SW stack and the cloud provides (virtual) HW.
  - PaaS – The cloud hides the lower layer SW and provides an application container (e.g. Flask) with "Your code goes here." You are aware of container.
  - FaaS – The cloud provides the container, and you implement functions (corresponding to routes in REST).

Serverless and Function-as-a-Service



https://medium.com/@tanmayct/serverless-architecture-function-as-a-service-19e127b8c990

What is Serverless Good/Not Good For ... ...

Serverless is **good** for

short-running
stateless
event-driven

- Microservices
- Mobile Backends
- Bots, ML Inferencing
- IoT
- Modest Stream Processing
- Service integration

Serverless is **not good** for

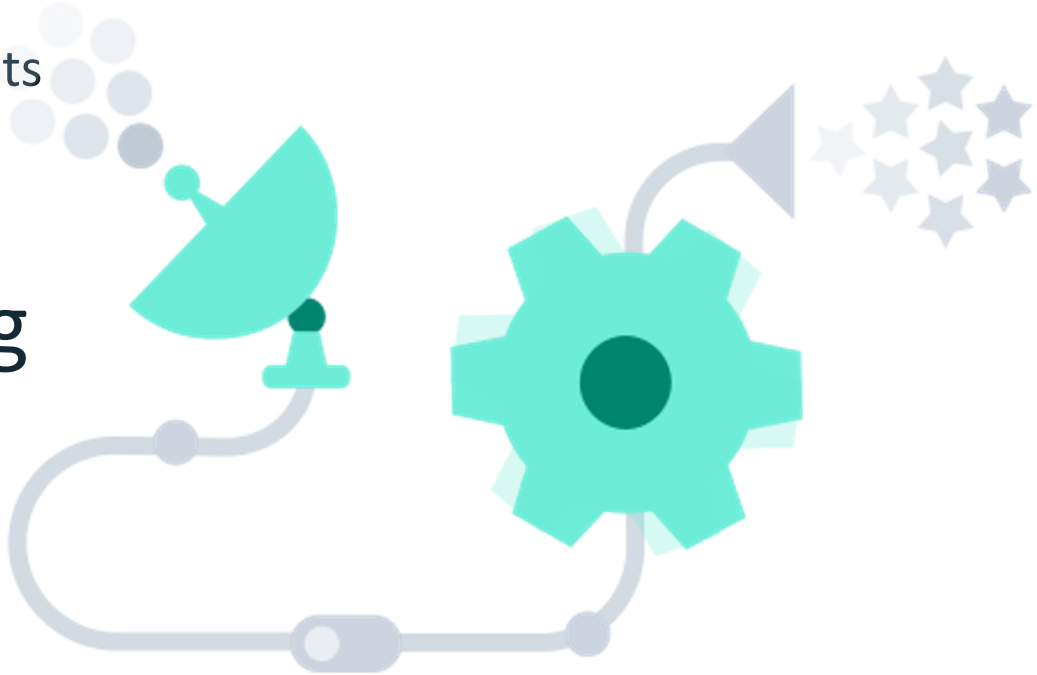
long-running
stateful
number crunching

- Databases
- Deep Learning Training
- Heavy-Duty Stream Analytics
- Numerical Simulation
- Video Streaming

What triggers code execution?

Runs code **in response** to events

Event-programming model



(Some) Current Platforms for Serverless

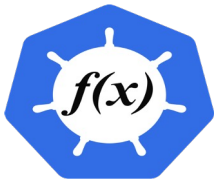
AWS Lambda

OpenLambda

Azure Functions

Kubernetes

IBM Cloud Functions



Red-Hat

Google Functions